14 | CPU视角：如何高效地利用CPU?

2021-04-14 吴磊

Spark性能调优实战

进入课程 >

讲述：吴磊

时长 22:22 大小 20.49M

你好，我是吴磊。

在日常的开发与调优工作中，总有同学向我抱怨：“为什么我的应用 CPU 利用率这么低？偌大的集群，CPU 利用率才 10%！”确实，较低的 CPU 利用率不仅对宝贵的硬件资源来说是一种非常大的浪费，也会让应用端到端的执行性能很难达到令人满意的效果。那么，在分布式应用开发中，我们到底该如何高效地利用 CPU？

我们说过，性能调优的最终目的，是在**所有参与计算的硬件资源之间寻求协同与平衡**，让硬件资源达到一种平衡、无瓶颈的状态。对于 CPU 来说，最需要协同和平衡的硬件资源，内存莫属。原因主要有两方面：一方面，在处理延迟方面，只有内存能望其项背；另一方面，在主板上内存通过数据总线直接向 CPU 寄存器供给数据。因此，理顺它们之间的关系，可以为性能调优奠定更好的基础。

那么，今天这一讲，我们就从硬件资源平衡的角度入手，去分析 CPU 与内存到底该如何合作。

CPU 与内存的平衡本质上是什么？

我们知道，Spark 将内存分成了 Execution Memory 和 Storage Memory 两类，分别用于分布式任务执行和 RDD 缓存。其中，RDD 缓存虽然最终占用的是 Storage Memory，但在 RDD 展开（Unroll）之前，计算任务消耗的还是 Execution Memory。**因此，Spark 中 CPU 与内存的平衡，其实就是 CPU 与执行内存之间的协同与配比。**

要想平衡 CPU 与执行内存之间的协同和配比，我们需要使用 3 类配置参数，它们分别控制着并行度、执行内存大小和集群的并行计算能力。只有它们设置得当，CPU 和执行内存才能同时得到充分利用。否则 CPU 与执行内存之间的平衡就会被打破，要么 CPU 工作不饱和，要么 OOM 内存溢出。

想要知道这 3 类参数都包含哪些具体的配置项，以及它们到底是怎么配置的，我们需要先弄清楚一些基础知识，也就是并行计算的线程之间是如何瓜分执行内存的。为了帮助你理解，我先来给你讲个故事。

黄小乙的如意算盘：并行计算的线程如何瓜分执行内存？

还记得地主招租的故事吗？与张麻子签订占地协议之后，黄小乙就开始盘算，自己分得的那块田地怎么经营才最划算。

他心想：“这么一大块地，我亲自种肯定划不来。一，我没有张麻子勤快；二，不管是种棉花还是咖啡都很耗时、费力，面朝黄土背朝天，我可耽误不起那功夫！不如，我把土地转让出去，让别人来种，我只管收购、销售，赚到的差价也够我吃穿不愁了！”

于是，他打定主意，贴出了一张告示。

本人黄小乙，洼子村人士。现有良田500顷，因本人能力有限，固将土地转让给有志之士。土地转让完全免费，但只面向方圆百里的父老乡亲，且转让的土地只能用于种植棉花或是咖啡，不得用作他用。棉花和咖啡丰收之后，一律由本人收购，收购标准为：

- 不符合质量要求者，承让人按市场价的2倍溢价倒赔本人损失
- 符合质量要求者，收购价格较本地市场价溢价10%

例如，棉农老王免费承让1亩地，丰收500斤棉花。假设500斤棉花全部符合质量要求，我将以市场价 * 1.1 * 500的价钱收购老王的棉花。相反，假设500斤棉花都不达标，老王将倒赔我市场价 * 2 * 500的损失。因此，望您好好珍惜免费承让的土地，用心地种植这些经济作物，可以说您是无本万利。相反，如果您尸位素餐，同样也会受到高额倒赔的惩罚。

告示日期：辛丑年农历二月初八

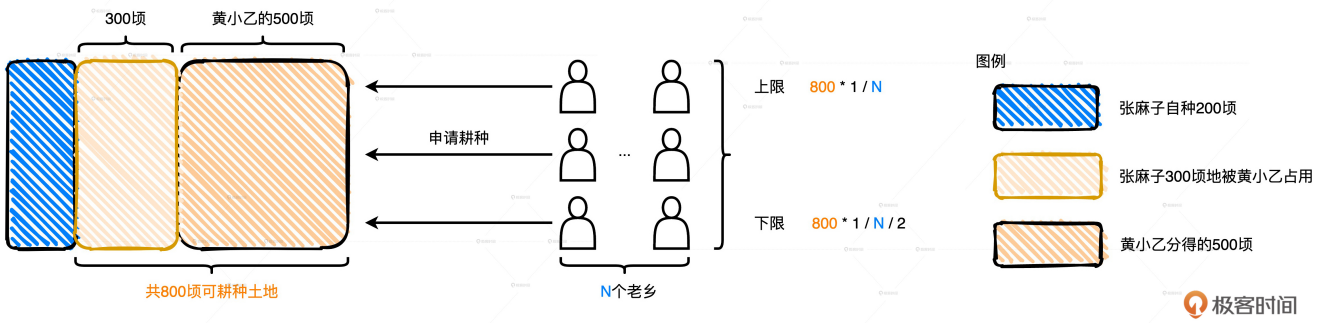
极客时间

黄小乙的告示

告示贴出去不到三天，十里八村的人都赶来承让土地，他们大部分都是吃苦耐劳的庄稼汉，一心想凭借这次机会改善生活，所以每个人恨不能把 500 顷的田地全都承让过来。

黄小乙见状，心中大喜，认为不仅自己的土地很快就可以被种满，还能名正言顺地去抢占张麻子的那块地。不过，也不能光图规模，还要确保棉花、咖啡的产出质量，更重要的是得想个办法让这种运作模式可持续。

于是，黄小乙追加了一项补充条款：“鉴于老乡们参与热情高涨，公平起见，我又制定了新的土地转让规则：**首先，每位老乡能够获得的土地面积有上下限，它的具体数值由可耕种土地总面积和申请人数共同决定；其次，土地转让权的有效时间与农作物生长周期一致，一旦作物丰收，承让人需让出土地，如有意愿再次耕种需重新申请。**”



土地转让规则

比如说，现阶段可耕种土地总面积已由 500 顷扩张为 800 顷（这是黄小乙就抢占了张麻子的地之后的土地总面积），如果有 400 位老乡申请土地转让权，那么每位老乡最高可得 2 顷（800/400）的土地，最低可得 1 顷（800/400/2）土地。也就是说，如果老乡人数为 N，那么每位老乡能够获得的土地面积会在（1/N/2，1/N）之间浮动。

这个规定大伙儿都心服口服，没过多久，800 顷土地就全部转让完了。一笔多赢的买卖让大伙都能各取所需，也让老谋深算的黄四郎都不禁心挑大指，感叹道“真是长江水后浪催前浪，一代新人换旧人！”

好啦，故事到这里暂时告一段落，但是黄小乙这份如意算盘和今天要讲的内容有什么关系呢？

我们讲过，黄小乙租赁的土地类比的是内存区域中的 Execution Memory。**在今天的故事里，黄小乙招募的棉农和咖啡农对应的就是，Executor 线程池中一个又一个执行分布式任务的线程。土地出让规则对应的就是，任务并发过程中多个线程抢占内存资源时需要遵循的基本逻辑。**

那么，执行内存抢占规则就是，在同一个 Executor 中，当有多个（记为 N）线程尝试抢占执行内存时，需要遵循 2 条基本原则：

执行内存总大小（记为 M）为两部分之和，一部分是 Execution Memory 初始大小，另一部分是 Storage Memory 剩余空间

每个线程分到的可用内存有一定的上下限，下限是 M/N/2，上限是 M/N，也就是均值

三足鼎立：并行度、并发度与执行内存

理清了线程与执行内存的关系之后，我们再来说说与并发度、执行内存和并行度这三者对应的 3 类配置项分别是什么，以及它们如何影响 CPU 与计算内存之间的平衡。

3 类配置项

我们讲过，并行度指的是为了实现分布式计算，分布式数据集被划分出来的份数。**并行度明确了数据划分的粒度：并行度越高，数据的粒度越细，数据分片越多，数据越分散。**

并行度可以通过两个参数来设置，**分别是 spark.default.parallelism 和 spark.sql.shuffle.partitions**。前者用于设置 RDD 的默认并行度，后者在 Spark SQL 开发框架下，指定了 Shuffle Reduce 阶段默认的并行度。

那什么是并发度呢？我们在配置项那一讲提到过，Executor 的线程池大小由参数 spark.executor.cores 决定，每个任务在执行期间需要消耗的线程数由 spark.task.cpus 配置项给定。两者相除得到的商就是并发度，也就是同一时间内，一个 Executor 内部可以同时运行的最大任务数量。又因为，spark.task.cpus 默认数值为 1，并且通常不需要调整，所以，**并发度基本由 spark.executor.cores 参数敲定**。

就 Executor 的线程池来说，尽管线程本身可以复用，但每个线程在同一时间只能计算一个任务，每个任务负责处理一个数据分片。因此，**在运行时，线程、任务与分区是一一对应的关系**。

分布式任务由 Driver 分发到 Executor 后，Executor 将 Task 封装为 TaskRunner，然后将其交给可回收缓存线程池（newCachedThreadPool）。线程池中的线程领取到 TaskRunner 之后，向 Execution Memory 申请内存，然后开始执行任务。

如果我们把棉农、咖啡农类比 CPU 线程，那么 TaskRunner 就可以理解为锄具，Task 要处理的数据分片可以理解为作物种子。有了锄具和种子之后，老乡们得去黄小乙那儿申请块地，才能开始耕种。

最后，我们再来说说执行内存。黄小乙的地就是执行内存，堆内执行内存的初始值由很多参数共同决定，具体的计算公式是：spark.executor.memory * spark.memory.fraction * (1 - spark.memory.storageFraction)。相比之下，堆外执行内存的计算稍微简单一些：spark.memory.offHeap.size * (1 - spark.memory.storageFraction)。

除此之外，在统一内存管理模式下，在 Storage Memory 没有被 RDD 缓存占满的情况下，执行任务可以动态地抢占 Storage Memory。因此，在计算可用于分配给执行任务的内存总量时，还要把有希望抢占过来的这部分内存空间考虑进来。这也是为什么黄小乙的可耕种土地总面积，会从最开始的 500 顷逐渐扩展到 800 顷。

由此可见，**可分配的执行内存总量会随着缓存任务和执行任务的此消彼长，而动态变化。但无论怎么变，可用的执行内存总量，都不会低于配置项设定的初始值**。

好啦，搞明白并行度、并发度和执行内存的概念，以及各自的配置项之后，我们再通过两个经常影响 CPU 利用率的例子，来说说它们是怎么影响 CPU 与计算内存之间的平衡的，由此总结出提升 CPU 利用率的办法。

CPU 低效原因之一：线程挂起

在给定执行内存总量 M 和线程总数 N 的情况下，为了保证每个线程都有机会拿到适量的内存去处理数据，Spark 用 HashMap 数据结构，以（Key，Value）的方式来记录每个线程消耗的内存大小，并确保所有的 Value 值都不超过 M/N。在一些极端情况下，有些线程申请不到所需的内存空间，能拿到的内存合计还不到 M/N/2。这个时候，Spark 就会把线程挂起，直到其他线程释放了足够的内存空间为止。

你可能会问：“既然每个线程能拿到的内存上限是 M/N，也就是内存总量对线程数取平均值，为什么还会出现有的线程连 M/N/2 都拿不到呢？这在数学上也不成立呀！”这是个好问题。这种情况的出现，源于 3 方面的变化和作用：

动态变化的执行内存总量 M

动态变化的并发度 N~

分布式数据集的数据分布

首先，动态变化的执行内存总量 M 我们刚刚已经说过了。M 的下限是 Execution Memory 初始值，上限是 spark.executor.memory * spark.memory.fraction 划定的所有内存区域。在应用刚刚开始执行的时候，M 的取值就是这个上限，但随着 RDD 缓存逐渐填充 Storage Memory，M 的取值也会跟着回撤。

另外，到目前为止，（1/N/2，1/N）上下限的计算我们用的都是线程总数 N，线程总数 N 是固定的。N 的取值含义是一个 Executor 内最大的并发度，更严格的计算公式是 spark.executor.cores 除以 spark.task.cpus。但实际上，上下限公式的计算用的不是 N，而是 N~。N~ 的含义是 Executor 内当前的并发度，也就是 Executor 中当前并行执行的任务数。显然 N~ <= N。

换句话说，尽管一个 Executor 中有 N 个 CPU 线程，但这 N 个线程不一定都在干活。在 Spark 任务调度的过程中，这 N 个线程不见得能同时拿到分布式任务，所以先拿到任务的线程就有机会申请到更多的内存。在某些极端的情况下，后拿到任务的线程甚至连一寸内

存都申请不到。不过，随着任务执行和任务调度的推进，N~ 会迅速地趋近于 N，CPU 线程挂起和内存分配的情况也会逐渐得到改善。

就像黄小乙的补充条款中举的那个例子一样，当可耕种土地总面积为 800 顷的时候，如果有 400 位老乡申请土地转让权，那么每位老乡最多可得 800/400=2 顷土地，最低可得 800/400/2=1 顷土地。

但如果这 400 位老乡不是同时来的，而是分两批来的，每批来 200 人的话，就会出现问题。按照他的规则，先来的这 200 位老乡，每人最多可得 800/200 = 4 顷土地。咱们前面说了，每个申请的老乡都想通过这次机会发点小财，于是这 200 位老乡每人都申请了 4 顷地，黄小乙的地一下子就被分光了！后来的 200 位老乡就没地可种了，他们只能等到第一批老乡的棉花和咖啡丰收了，再重新一起申请土地转让权。

假设第一批老乡同时大丰收，按照黄小乙转让规则的第一条，第一批老乡要交出土地使用权，如果想要继续耕种的话，就得和第二批老乡一起重新申请。在这种情况下，上下限的计算才是黄小乙最开始举例的那种算法。

第三个影响任务并发度和内存分配的因素，是分布式数据集的分布情况。在刚才的例子中，如果第一批老乡每人只申请 2 顷土地，那么第二批老乡来了之后依然有地可种。每人申请多大的土地，取决于他手里有多少农作物种子，我们之前把每个 Task 需要处理的数据分片比作是作物种子，那么，数据分片的数据量决定了执行任务需要申请多少内存。**如果分布式数据集的并行度设置得当，因任务调度滞后而导致的线程挂起问题就会得到缓解。**

CPU 低效原因之二：调度开销

线程挂起的问题得到缓解，CPU 利用率就会有所改善。既然如此，是不是把并行度设置到最大，每个数据分片就都能足够小，小到每个 CPU 线程都能申请到内存，线程不再挂起就万事大吉了呢？

当然不是，并行度足够大，确实会让数据分片更分散、数据粒度更细，因此，每个执行任务所需消耗的内存更少。**但是，数据过于分散会带来严重的副作用：调度开销骤增。**

对于每一个分布式任务，Dirver 会将其封装为 TaskDescription，然后分发给各个 Executor。TaskDescription 包含着与任务运行有关的所有信息，如任务 ID、尝试 ID、要处理的数据分片 ID、开发者添加的本地文件和 Jar 包、任务属性、序列化的任务代码等

等。Executor 接收到 TaskDescription 之后，首先需要对 TaskDescription 反序列化才能读取任务信息，然后将任务代码再反序列化得到可执行代码，最后再结合其他任务信息创建 TaskRunner。

因此你看，每个任务的调度与执行都需要 Executor 消耗 CPU 去执行上述一系列的操作步骤。数据分片与线程、执行任务一一对应，**当数据过于分散，分布式任务数量会大幅增加，但每个任务需要处理的数据量却少之又少，就 CPU 消耗来说，相比花在数据处理上的比例，任务调度上的开销几乎与之分庭抗礼。**显然，在这种情况下，CPU 的有效利用率也是极低的。

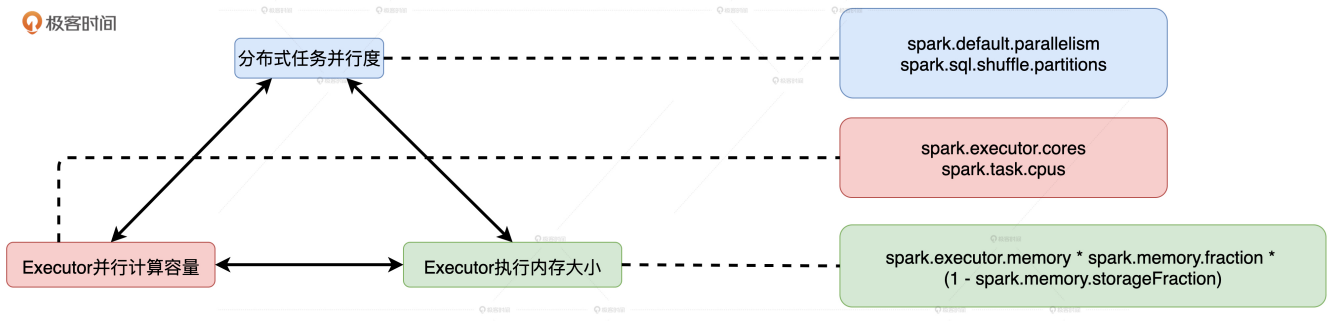
如何优化 CPU 利用率?

你可能会说：“这也太尴尬了，并行度低了不行，容易让 CPU 线程挂起；高了也不行，调度开销太大，CPU 有效利用率也不高。高也不行、低也不行，那我该怎么办呢？”

咱们不妨来算笔账。我们还是拿黄小乙的如意算盘来举例，如果 400 个老乡同时来申请他的 800 顷地，那么每个老乡能分到 1 到 2 顷土地不等。相应地，每位老乡需要购买的种子应该刚好够种满 1 到 2 顷地。因为，买多了种不下，买少了还亏。假设洼子村农产品交易市场的种子总量刚好够种 1000 顷地，从卖家的视角出发，这些种子应该售卖 1000/2 =500 到 1000/1 = 1000 次，才能赚到最多的钱。

因此，在给定 Executor 线程池和执行内存大小的时候，我们可以参考上面的算法，**去计算一个能够让数据分片平均大小在（M/N/2, M/N）之间的并行度，这往往是个不错的选择。**

总的来说，对 CPU 利用率来说，并行度、并发度与执行内存的关系就好像是一尊盛满沸水的三足鼎，三足齐平则万事大吉，但凡哪一方瘸腿儿，鼎内的沸水就会倾出伤及无辜。



三足鼎立

小结

今天这一讲，我们从 CPU 与执行内存平衡的角度，通过梳理 Executor 并行度、并发度和执行内存之间的关系，以及它们对 CPU 利用率的影响，总结出了有效提升 CPU 利用率的方法。

首先，在一个 Executor 中，每个 CPU 线程能够申请到的内存比例是有上下限的，最高不超过 1/N，最低不少于 1/N/2，其中 N 代表线程池大小。

其次，在给定线程池大小和执行内存的时候，并行度较低、数据分片较大容易导致 CPU 线程挂起，线程频繁挂起不利于提升 CPU 利用率，而并行度过高、数据过于分散会让调度开销更显著，也利于提升 CPU 利用率。

最后，在给定执行内存 M、线程池大小 N 和数据总量 D 的时候，想要有效地提升 CPU 利用率，我们就要计算出最佳并行度 P，计算方法是让数据分片的平均大小 D/P 坐落在（M/N/2, M/N）区间。这样，在运行时，我们的 CPU 利用率往往不会太差。

每日一练

1. 从 Executor 并发度、执行内存大小和分布式任务并行度出发，你认为在什么情况下会出现 OOM 的问题？
2. 由于执行内存总量 M 是动态变化的，并发任务数 N~ 也是动态变化的，因此每个线程申请内存的上下限也是动态调整的，你知道这个调整周期以什么为准？

期待在留言区看到你的思考和答案，如果你的朋友也在为提高 CPU 利用率苦恼，欢迎你把这一讲转发给他，我们下一讲见！

提建议

更多课程推荐

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 13 | 广播变量（二）：有哪些途径让Spark SQL选择Broadcast Joins?

下一篇 15 | 内存视角（一）：如何最大化内存的使用效率?

精选留言 (13)

写留言

qconljk
2021-04-14

首先，在一个 Executor 中，每个 CPU 线程能够申请到的内存比例是有上下限的，最高不超过 1/N，最低不少于 1/N/2，其中 N 代表线程池大小。这个除以2，2代表什么?

展开

作者回复: 好问题，其实没有什么特别的含义，就是一种公平机制，就是保证至少有均值的1/2可以满足，否则就不进行计算。

其实你说把它改成3成不成，我觉得也没什么不可以。但是，改成3之后，task最低内存保障更低了，即便你有1/3给它，它也完不成计算，其实还是得挂起。

1/2的保底，其实更make sense，因为1/2的内存相比均值来说，只差了一半，这样有些任务你

先分配了1/2，运行的过程中，其他task还会释放内存，这个时候，这个task还是可以成功执行的。但如果是1/3，你亏空的内存更多，需要等待的概率越大，挂起的概率也就越大。

2 4

斯盖丸
2021-04-14

这讲看得有些迷，想请问下老师如何从UI角度看出来我任务的并行度是否合适呢？

展开

作者回复: Spark UI有专门的配置页，记录了不同配置项的设置值，其中有默认并行度设置。

但重点不在这，重点是协调、平衡“三足鼎立”，也就是并行度、并发度、执行内存，去提升CPU利用率，所以你更需要使用系统工具、监控工具，比如ganglia、Prometheus、Grafana、nmon、htop等等，这些工具，去观察你的CPU利用率，然后回过头来，平衡三者，然后再去观察CPU利用率是不是提升了，通过这种方式，来去调优。

3

zxk
2021-04-17

问题一：并发度决定了数据分片的大小：
- 在每个线程都分配到了最大内存，即 M/N 的内存时，如果 task 还需要更多的内存，那么就会发生 OOM。
- 在每个线程都分配到了最少内存，即 M/2N的内存时，如果 task 还需要更多的内存，此时又没有其他线程释放内存供其使用，那么也会导致OOM。

展开

作者回复: 没问题，满分💯 ~

2

Geek_d794f8
2021-04-16

老师，在考虑并行度,内存,线程数三者之间的平衡时，spark.sql.shuffle.partitions的值是shuffle的reducer阶段的并行度，那么对于从数据源读取(比如读hive表)这个起始的map阶段的并行度是否需要考虑？这个阶段spark底层有某种默认的切片规则吗，需要在代码中人为的干预吗(比如coalesce)？ 我使用的是DataFrame和DataSet的api。

展开

作者回复: 都需要考虑，结合你这个例子，整体逻辑是这样的。

首先，Spark读取分布式文件，获取数据源，这个时候，并行度就是文件在分布式文件系统上的并行度，比如HDFS、比如S3。HDFS的分片可能是128M或是256M，那么它的并行度，就取决于文件总大小和分片大小的商。

这个时候，由于分片大小是固定的，你可以结合分片大小，去设置执行内存和并发度（executor线程池），让他们满足（1/N/2，1/N）的关系。

然后，你设置的spark.sql.shuffle.partitions，会控制Joins之中的Shuffle Reduce阶段并行度，这个参数设置多少。其实取决于你Shuffle过后每个Reduce Task需要处理的数据分片大小。由于你之前是按照128M或是256M来设置的执行内存， 和并发度。这个时候，在设置spark.sql.shuffle.partitions这个值的时候，只要保证数据分片大小还是在128M或是256M左右（shuffle前可能有过滤、过程中还会有聚合，所以原来的并行度就不合适了），就依然能维持“三足鼎立”的计算平衡。

所以说，核心是维持这个平衡。在你这个case，核心思路是，根据“定下来的”，去调整“未定下来的”，就可以去设置每一个参数了。

在最开始，“定下来的”是并行度，因为这个是文件系统决定的，但是执行内存和并发度未定，所以你可以相应地去调整这两类参数。

后面Shuffle的时候，执行内存和并发度已经定下来了，但是spark.sql.shuffle.partitions未定，你再结合那个公式，去定这个参数的值就好了。思路就是这么个思路，其实还是万变不离其宗。

2

Geek_d794f8

2021-04-16

我的理解是每个线程申请的内存上限是M/N,那么当数据分片过少，某个task需要处理的数据量较大，M/N的上限执行内存也不够时，就会出现OOM。
不知道这么理解对不对?

展开

作者回复: 对，没错~

2

来世愿做友人 A

2021-04-16

1. excutor 并发度如果过高，考虑极端情况，storageMemery 还没有使用，这时候这部分内存是会被 task 所抢占的，每个任务的上限是 1/N。但是对于 storage 和 execute pool 的池锁不是同一个锁。并发高的情况下，可能会出现 execute 剩余内存假设 1M，task 申请 execute 内存并且 size 申请成功，但是还没 alloc 内存。并行条件下，此时别的 task 线程刚好申请了 storage pool 的内存 0.8M 进行 block 存储，并且申请成功。然后申...

展开

作者回复: 第二题完美~

第一题也对，不过算是edge case了。可以想想，一般情况下，in general，为啥task会OOM?

1 1

kingcall

2021-04-14

M 的下限是 Execution Memory 初始值，上限是 spark.executor.memory * spark.memory.fraction 划定的所有内存区域。这个老师笔误，写错了吧！

展开

作者回复: 没写错哟，上限就是 spark.executor.memory * spark.memory.fraction ，也就是storage memory + execution memory之和。Unified memory manager，在统一管理模式下，大家是可以互相抢占的，因此，如果没有分布式数据集缓存，storage memory那片内存区域，执行任务是都可以抢过来的，所以上限就是两者之和。

2 1

aof

2021-05-04

每个线程申请执行内存的时候都会跟所请求的内存大小进行比较。为一个内存消费者（MemoryConsumer）申请执行内存的具体实现逻辑在TaskMemoryManager#acquireExecutionMemory()方法中，这个方法为内存消费者（consumer）申请指定大小的内存空间，如果没有足够的内存，将会对consumer进行spill()来释放更多内存，具体要对哪些consumer进行spill()，会有一个排序算法（使用了TreeMap）。

展开

小灵芝

2021-04-30

“在给定执行内存 M、线程池大小 N 和数据总量 D 的时候，想要有效地提升 CPU 利用率，我们就要计算出最佳并行度 P，计算方法是让数据分片的平均大小 D/P 坐落在（M/N/2, M/N）区间。这样，在运行时，我们的 CPU 利用率往往不会太差。”

请问老师，这里的M, N， D 都是针对一个executor而言的对吧?

展开

作者回复: 稍有差别，M、N是针对Executor的，也就是Executor的执行内存M、线程池大小N。

D和P不一样，它指的是你的分布式数据集，D是数据总量，比如20GB；而P指的是这份数据集的并行度，比如200，那么你的每个数据分片的大小D/P，就是20GB/200 = 100MB。

如果你的M是2GB，也就是2GB的执行内存，N是20，也就是20个线程，那么这个时候，M/N就是100MB。那么，你的D/P就刚好坐落在(M/N/2, M/N)这个区间里。

小灵芝
2021-04-30

“并行度可以通过两个参数来设置，分别是 spark.default.parallelism 和 spark.sql.shuffle.partitions。前者用于设置 RDD 的默认并行度，后者在 Spark SQL 开发框架下，指定了 Shuffle Reduce 阶段默认的并行度。”

请问老师，这两个参数是都要设置吗？还是说在用RDD的时候设置前者，Dataframe或...

展开

作者回复: 对，看你的应用使用哪个API，RDD的话，用前者；DF或是DS的话，用后者。

Fendora范东_
2021-04-15

1.当executor内所有线程都在运行，实际需要内存比预估申请内存大，这个时候执行内存又不能继续扩展，就会出现oom
2.executor内正在运行的这批任务执行完，下一批任务被执行前，就进行资源调整。根据此时执行内存大小/min(待分配任务数，executor.cores)进行内存分配

展开

作者回复: 第二题答得很完美~ 💯

第一题也对。不过，结合（1/N/2，1/N）的上下限，能再往细了说一说吗?

2

王天雨

2021-04-14

请问M 和 N 分别对应的是单个Executor的内存和核数，还是针对某个spark任务分配的总Executor数量和核数呢

展开

作者回复: 单个Executor的内存和核数，M和N，都是一个Executor内的范畴，方便理解和讨论~

Fendora范东_

2021-04-14

磊哥，每个线程分配内存下限 我理解应该是M/2N呢?

展开

作者回复: 是啊，你说的没错，M/2N，M/N/2，一个意思哈~